

Blazing Saddles: Getting the Performance Out of VCS

Gregg D. Lahti
Corrent Corporation
gregg.lahti@corrent.com

Tim Schneider
Synopsys Corporation
tim.schneider@synopsys.com

Gopal Varshney
Corrent Corporation
gopal.varshney@corrent.com

ABSTRACT

It's three days before your tapeout deadline and an RTL change from above gets dropped into your design. The RTL and gate-level simulation regressions need to be re-run across the compute ranch. You're now under the gun to get it completed to make the tapeout or you may be the next gateslinger in the manager's layoff sights at high noon. Are you sure you're getting the most performance out of your VCS simulation tool?

The huge RTL and gate-level simulation burden for verifying multi-million gate designs can make or break a project deadline. This paper will discuss methods, tips and tricks that proved successful at Corrent of getting the most performance out of your VCS simulations so you too can have the fastest VCS simulation in the West.

1.0 Introduction

VCS is a fast Verilog simulator provided you know the right switches and code your Verilog to make use of the built-in performance. VCS has been continually enhanced over the years, and the myriad of command-line switches can make the usage a bit unmanageable at times. VCS is generally quite fast out of the box, but the compiler solves for the “general” case. Using knowledge from the designer or verification engineer, VCS can be coaxed into more efficiently simulating Verilog code. The right command line switches thrown for the right reasons can make all the difference in the performance of your simulation runs.

2.0 Debugging vs. Performance

VCS should be viewed as having two different operating models: debugging and regression. Debugging options within VCS add more visibility into the design at the cost of slowing the simulation down, and VCS will show its true speedy colors without the debugging overhead. So which one do you use?

2.1 Debugging with VCS

The time-spent usage model says that roughly 75% of an engineer’s time is verifying the design functionality with about 25% or less of an engineer’s time spent coding¹. Considering how much time an engineer spends debugging, make VCS work more efficiently for debugging by selecting only the compile and run-time options necessary for that simulation. Adding PLI’s for waveform debugging will slow VCS down, but there are still many options that can be modified for a speedy sim and still allow for debugging information to be kept during the simulation.

2.2 Speedy VCS Simulation for Regressions

Once the design is near completion and system simulation is underway, regression simulations are generally used on a frequent basis to test out the design. Regressions are generally a large suite of tests that perform functionality tests across the system. Usually these regression simulations are large in scope, large in numbers, and long in wall-clock time. At Corrent, we use a regression suite of over 2000 tests to run against our Socket ArmorTM and Packet ArmorTM ASICs test benches across a compute ranch of 50 high performance Linux machines. Due to the number of scenarios, it was very important to strip out all of the debugging features and make these sims run extremely fast. Our unoptimized regression suite took an average of 3 weeks of wall-clock time to run. After optimization, this time was cut dramatically to just under 1 week!

3.0 Items that Make VCS Slow

The last thing an engineer wants is to wait around for a simulation to finish up. It’s a great excuse to get coffee, but productivity slides as the simulations and gate count get exponentially larger. When you want VCS to run fast, here’s what *not* to do.

¹ Test Benches: The Dark Side of IP Reuse, Gregg D. Lahti, San Jose SNUG 2000 paper.

3.1 VCS Command Line Switches

The first item to optimization is look for command-line switches in your compile script that are time-sinks and CPU hogs. Here's a quick list of flags to be aware of.

- `-I`
- `-PP`
- `+cli`
- `+acc+2`
- missing `-Mupdate`
- `-P [library]`

Many times compile scripts get written quickly to get the simulation up and functional but weren't proofed for basic things that may not ever get used or needed except in rare circumstances. A cursory check of the compile and run-time command line flags can optimize your simulation without affecting the simulation usage.

3.1.1 The `-I` and `-PP` Flags

The `-I` flag allows interactive debug GUI simulation ability in VCS. This feature is great to use for tracing those rare race conditions that are hanging a simulation or causing mismatches. However, the `-I` switch is a complete time-sink in performance, as the simulator must remember state information across all registers and nets instead of optimizing and collapsing redundant/similar logic. Be sure to check scripts for this flag, as sometimes the `-I` flag meant for a C compiler include gets separated and interpreted as an interactive flag instead. If the user wants to enable the simulation to write a compact dumpfile for post process debug, the usage is to insert a `$vcdpluson` call in the design (just like `$dumpvars`) and compile VCS with `-PP` to enable dumping for post processing. Note that the `-PP` flag is also a time sink due to the extra effort of saving state information. If debugging isn't required, remove the `-I` and `-PP` flags.

3.1.2 The `+cli` Flag

The `+cli` flag is another time-sink for simulation since it globally enables the interactive command line interpreter which is a set of interactive commands that can be used to traverse the design and probe signal value and drivers. It's a great feature to use when in conjunction with the `-I` flag for tracing race conditions, but almost never used in the "run sim and then debug" model of debugging.

3.1.3 Absent `-Mupdate` Flag

Subsequent compiles can benefit with the `-Mupdate` flag. This allows VCS to update only the portions of Verilog code that have changed rather than compile the entire design, saving time.

3.1.4 The `+acc+2` Switch

This flag is another performance inhibitor for most simulations. This flag is an old obsolete way to enable visibility within the design for PLI applications. A better method is to enable these capabilities within the `-P` file.

3.1.5 -P [pli dispatch table] Flags

Make sure that your pli.tab files (dispatch table for pli calls) is optimized. The key item to keep in mind is to ensure that you are not enabling more PLI visibility and capabilities than are really needed. The pli.tab file optimization will be covered later in this paper, plus there is a whole section in the VCS reference manual that discusses this in detail.

3.2 Coding Styles

Coding styles play an important part to how efficiently the simulation can be run. Good coding practices result in good simulation, good synthesis, and a correct ASIC. Cliff Cummings has a great course on Verilog coding principles and his website <http://www.sunburst-design.com> is loaded with SNUG papers he has written on coding styles. Here are a few basic items to watch out for while you wait to take his class.

3.2.1 Addition of # Delays

Delay loops, delay chains, and other timing additions to a simulation will slow a simulation down. VCS cannot use cycle-based optimizations in the simulation if #1 delays are inserted on the Right Hand-Side of the assignment, such as delaying the Q output of a register element by 1 time element:

```
always @(posedge clk or negedge reset_n) begin
    if (~reset_n) q <= #1 0;
    else      q <= #1 d;
end // always
```

Figure 1.0 Inefficient Verilog Coding Using Delay Assignments

VCS unnecessarily schedules the q assignment one delay element past the clock transition causing the simulation to run slower. Removing the #1 or #0 delay assignments in your code can increase the simulation performance from 0 to 200% with an average of 30-50% increase².

3.2.2 Use of initial and always Blocks

The following construct is legal Verilog, but not legal to use in good coding practices:

```
always begin
    mysig <= 1;
end // always
```

Figure 2.0 Bad Verilog Coding Using Always Block

In Figure 2.0 an initial block should have been used instead of an always block. VCS may or may not optimize this, and your simulations that worked for days may suddenly not work and develop infinite loop conditions if this type of coding is used. Instead, change the always to initial.

² Verilog Nonblocking Assignments With Delays, Myths and Mysteries, Cliff Cummings, Boston SNUG 2002 paper.

3.2.3 Mixing blocking and non-blocking assignments

The VCS compiler doesn't treat combinatorial and sequential logic the same. Because of this, some combinatorial items may get optimized differently if it is grouped. By mixing block and non-blocking assignments (essentially combinatorial elements and sequential elements) in Verilog always blocks may cause different simulation speed results. The VCS compiler can aggressively optimize combinatorial and sequential logic, but cannot be as affective if they are mixed together.

3.2.4 Grouping Non-Blocking Assignments

VCS does a good job of collapsing always blocks and coding style doesn't play a part in simulation performance when it comes having one always block containing multiple assignments or many always blocks with one assignment. For example, a Johnson counter could be written as the following two ways but simulation speed is essentially identical:

```
module jcntr (q1, q2, q3, q4, clk, rst_);
  output q1, q2, q3, q4;
  input  clk, rst_;

  reg q1, q2, q3, q4;
  wire clk, rst_;

  always @(posedge clk or negedge rst_) begin
    if (rst_ == 0) q1 <= 0;
    else          q1 <= ~q4;
  end
  always @(posedge clk or negedge rst_) begin
    if (rst_ == 0) q2 <= 0;
    else          q2 <= q1;
  end
  always @(posedge clk or negedge rst_) begin
    if (rst_ == 0) q3 <= 0;
    else          q3 <= q2;
  end
  always @(posedge clk or negedge rst_) begin
    if (rst_ == 0) q4 <= 0;
    else          q4 <= q3;
  end
end
endmodule
```

Figure 3.0 Assignments in Separate Always Blocks

```

module jcntr (q1, q2, q3, q4, clk, rst_);
  output  q1, q2, q3, q4;
  input   clk, rst_;

  reg  q1, q2, q3, q4;
  wire clk, rst_;

  always @(posedge clk or negedge rst_) begin
    if( rst_ == 0) begin
      q1 <= 0;
      q2 <= 0;
      q3 <= 0;
      q4 <= 0;
    end
    else begin
      q1 <= ~q4;
      q4 <= q3;
      q2 <= q1;
      q3 <= q2;
    end
  end
endmodule

```

Figure 4.0 No Grouping Of Assignments into One Always Block

Either of these coding styles is acceptable, as VCS collapses the logic for performance.

3.3 PLI and .tab Files

Due to limitations in the Verilog language, many engineers use PLI's to add C-based functionality to a test bench or simulation. However, the PLI structure with VCS is inherently slow because VCS is a compiler technology. It is best at generating fast efficient direct object code from Verilog. External C calls using the PLI interface are just that, "external". VCS has no clue at compile time what is in that code, and what placeholders are required for an external pli task to talk with Verilog. The pli.tab file is the means to this end. Depending upon what is being done with the external PLI, routines can require different kinds of design access. Perhaps a pli task is only reading values and not driving the design. This would only call for 'read' access and the task should be flagged as only needing such in the pli.tab file. There are several different levels of abstraction in the Verilog pli interface. The lowest level calls are referred to as tf routines. Higher-level routines can be found in the list of acc pli calls, with the highest level of pli calls being vpi routines. All acc, pli and tf calls are supported in VCS. However, a faster, more efficient way to implement C into a Verilog design within VCS is to use the Direct C feature. This allows you to include C source code directly into your Verilog, and creates the most efficient native compiled code for a chosen platform. Since VCS knows the rules for Direct C, it will inline the C code directly into its native code for the design resulting in much lower overhead and higher performance.

3.4 Disk & Network I/O

An important item to remember is that disk I/O is 10X faster than network I/O. Large compilations that take hours can be optimized if the object files are written to local disk rather than a NFS filesystem. Use the `-o /tmp/simv -mdir /tmp/csrc` option to point the object files to be written to a local disk area (/tmp is usually sufficient providing it has enough disk space allocated). Bribing your friendly sysadmin for a temp data partition on your workstation can't hurt, either.

4.0 Items that Make VCS Quick

Now that we've discussed some of the pitfalls that slow VCS, let's look at some optimization techniques that will help the simulation speed and performance. These optimizations are geared for regression runs, where debugging information isn't required. These are the easy optimizations to make since it usually just requires a cursory review of the compile and run scripts used in the simulation environment.

4.1 VCS Command Line Switches

About the time Synopsys and Viewlogic merged, the Chronologic group had purchased a technology called Radiant. This small group of engineers had a Verilog preprocessor that optimized Verilog code for simulation performance. The Radiant technology was subsequently integrated into the VCS 5.0 release. Radiant technology optimizes the execution of Verilog code by as much as 2-5X in simulation speed³. Use of the `+rad` command at the compilation phase can in most cases speed up simulation time, although the actual speedup will vary according to coding style, design functionality, and PLI usage. Note that `+rad` is best used for RTL or non-timing gate level (`+nospecify`) simulations. It's a great flag to use!

Another option is the `+2state` flag to force the simulation to run in two-states instead of seven. This flag has implications on coding style and for the most part is less than useful, but can be worth a try if you don't require all the Verilog states. It is also useful to test functionality in your design without worrying about getting out of reset due to X propagation.

4.2 Coding Styles

Coding style does have an important part of how a simulation will work. Delay loops, delay chains, and other timing additions to a simulation will slow a simulation down. Don't code using examples from the **Items that Make VCS Slow** section of the paper.

4.3 In-Line C

If possible, re-code your PLI routines into in-line C (Direct C) format. This does limit your use of the code to VCS only, but hey, this is a VCS paper! ☺ The added ease of use and perf may be worth it.

5.0 Profiling Your Simulations

So how do you know if all of these command-line switches and coding techniques really work? The answer is found in the output of the `+prof` command line option. VCS will happily tell you where it is spending time in your Verilog code and provide enough statistical data to choke a horse. Here's how to get through the gory details.

Previous versions of VCS required a separate step to profile code execution using `gprof`. Profiling is a very easy procedure as of VCS 5.2 now that a profiling option has been included

³ VCS 5.0 release notes

into the compiler. Add the `+prof` in your compile script and then compile and run the simulation. The output will be found in the `vcs.prof` file.

5.1 First Analysis

Once the profile file is created after the simulation is complete, the fun begins. Let's take a profiling example from a basic start to an optimized finish and see how much performance can be gained.

Here is a diagrammed profile log of an un-optimized simulation.

```
//      Synopsys VCS 6.2R12
//      Simulation profile: vcs.prof
//      Simulation Time:      976.180 seconds

=====
                        TOP LEVEL VIEW
=====
                        TYPE          %Totaltime
-----
                        PLI           0.23
                        VCD           0.99
                        KERNEL        7.76
                        DESIGN        91.02
-----

=====
                        MODULE VIEW
=====
Module(index)          %Totaltime   No of Instances   Definition
-----
delaychain             (1)      67.75           56      ../top/rtl/delaychain.v:15.
dll_delay_line        (2)      7.16            2      ../rtl/ddrctlr/rtl/dll_delay_line.v:21.
ckrst                 (3)      2.47            1      ../top/rtl/ckrst.v:13.
INVDL                 (4)      1.25           8431   /projects/clibs/umc/0.15vst/
tapeoutkit/stdcell/UMCL15U210T2_2.2/Verilog_simulation_models/INVDL.v:32.
hurricane_tb         (5)      1.23            1      ../tb/hurricane_tb.v:31.
pdisp                (6)      1.16            8      ../rtl/hurricane/rtl/pdisp.v:33.
dll_mux              (7)      0.96           1720   ../rtl/ddrctlr/rtl/dll_mux.v:21.
dll_buf              (8)      0.56           1732   ../rtl/ddrctlr/rtl/dll_buf.v:21.
spsram_1536x32       (9)      0.54            8      /projects/clibs/rams_nobist_M4one/
0.15vst_2.0/Verilog_fix/spsram_1536x32.v:8.
-----
```

Figure 5.0 Profile Log of Un-optimized Simulation

In figure 4.0, **Top Level View** section shows the total percentage of time the simulation pieces executed. In our case, 91% of the time, the design was being executed in VCS with 7.76% of the time VCS kernel operations were taking place. This quick summary is useful to see if the PLI is occupying a large percentage of simulation time compared to the actual design. The simulation time is pretty large. Upon closer inspection, the `-I` flag and `+acc+2` flag were set in the compile script. These timesinks gobble huge amounts of valuable VCS CPU time, so these first get removed from the compile script.

The **Module View** shows how much percentage of time each module of the design was occupying of the total design percentage time. This section of the report shows that VCS spent 67% of time in the one particular area, the `top/rtl/delaychain.v` file. That's a very

significant percentage of time compared to the rest of the design. Also, 7% of the time is spent in the `rtl/ddrctlr/rtl/dll_delay_line.v` file. Further investigation in the Verilog code is required to clean up these two areas.

5.2 Cleaning the Verilog Code

At first glance, almost 75% of the design simulation time is spent in two distinct areas. Both appear to be delay items in the simulation. Now that we know where the time is being spent, let's take a look at the Verilog code.

```
module delaychain (
    sigin,
    sigout
);

input sigin;
output [120:0] sigout;
wire [120:0] sigout;

BUFD1 buf_u000 (.A(sigin), .Z(sigout[0]));
INVDL inv_u001 (.A(sigin), .Z(sigout[1]));
INVDL inv_u002 (.A(sigout[1]), .Z(sigout[2]));
INVDL inv_u003 (.A(sigout[2]), .Z(sigout[3]));
INVDL inv_u004 (.A(sigout[3]), .Z(sigout[4]));
...
INVDL inv_u117 (.A(sigout[116]), .Z(sigout[117]));
INVDL inv_u118 (.A(sigout[117]), .Z(sigout[118]));
INVDL inv_u119 (.A(sigout[118]), .Z(sigout[119]));
INVDL inv_u120 (.A(sigout[119]), .Z(sigout[120]));

Endmodule
```

Figure 6.0 Delay Chain Verilog Code

A delay chain using a string of 120 inverters was implemented at the clock unit of the chip. This delay chain provided a programmable tap for the layout and back-end team to add or remove delay of the generated clocks, allowing hand-tuning of the clock insertion delay to the floor-planned top-level modules of the chip so that all of the clocks could be equalized. This method was used as a workaround due to Apollo's anemic clock tree insertion capabilities on a very large die in conjunction with the limitations of a good floorplanner. Clearly the RTL simulation doesn't require the delay chain for functional simulation, so we can optimize this code out. First, we tried using the `+nospecify` option as a quick fix to remove any delay timing in the cell. We also removed the huge timesink flags, `-I` and `+acc+2`, from the compile script to improve the simulation performance.

```
// Synopsys VCS 6.2R12
// Simulation profile: vcs.prof
// Simulation Time: 149.660 seconds
```

```
=====
TOP LEVEL VIEW
=====
```

TYPE	%Totaltime
PLI	1.65
VCD	0.02
KERNEL	11.86
DESIGN	86.46

```
-----
```

```
=====
MODULE VIEW
=====
```

Module(index)		%Totaltime	No of Instances	Definition
delaychain	(1)	17.24	56	../top/rtl/delaychain.v:15.
hurricane_tb	(2)	8.43	1	../tb/hurricane_tb_nodump.v:31.
dll_mux	(3)	5.06	1720	../rtl/ddrctlr/rtl/dll_mux.v:21.
INVDL	(4)	4.12	8431	/projects/clibs/umc/0.15vst/ tapeoutkit/stdcell/UMCL15U210T2_2.2/Verilog_simulation_models/INVDL.v:32.
pdisp	(5)	3.83	8	../rtl/hurricane/rtl/pdisp.v:33.
dll_buf	(6)	3.10	1732	../rtl/ddrctlr/rtl/dll_buf.v:21.
rctl	(7)	2.35	8	../rtl/hurricane/rtl/rctl.v:32.
dll_delay_element	(8)	2.24	1720	../rtl/ddrctlr/rtl/ dll_delay_element.v:20.
xaux_regs	(9)	1.60	8	../rtl/hurricane/rtl/xaux_regs.v:249.
tdc_cdb	(10)	1.00	1	../rtl/tdc/rtl/tdc_cdb.v:16.
dpsram_b_64x64	(11)	0.98	2	/projects/clibs/rams_nobist_M4one/ 0.15vst_2.0/Verilog_fix/dpsram_b_64x64.v:28.
cr_int	(12)	0.98	8	../rtl/hurricane/rtl/cr_int.v:287.
dpsram_b_160x64	(13)	0.97	1	/projects/clibs/rams_nobist_M4one/ 0.15vst_2.0/Verilog_fix/dpsram_b_160x64.v:28.
saob	(14)	0.93	8	../rtl/saob/rtl/saob.v:162.
mt46v16m16	(15)	0.88	10	../vmodels/mt46v16m16.v:48.

Figure 7.0 New Profile with +nospecify Option

Notice that the simulation time drastically decreased from 976 seconds down to 149.6 seconds! The `rtl/ddrctlr/rtl/dll_delay_line.v` item has moved down the CPU hog list since the `+nospecify` was added, so we have appeared to clean up that area a bit. However, the `delaychain` item is still in the top list of CPU hogs. Since we know that this is a gate-level workaround coded in RTL, we can optimize the code further.

A better approach to the delay chain would be to completely remove the delay chain with ``ifdef` options for gate-level simulation only. Again, this is a Verilog coding style issue that has a drastic affect on speed. Here is the modified `delaychain.v` code as shown below:

```

module delaychain (
    sigin,
    sigout
);

input sigin;
output [120:0] sigout;
wire [120:0] sigout;

`ifdef SYNTH_DELAYCHAIN

BUFD1 buf_u000 (.A(sigin), .Z(sigout[0]));
INVDL inv_u001 (.A(sigin), .Z(sigout[1]));
INVDL inv_u002 (.A(sigout[1]), .Z(sigout[2]));
INVDL inv_u003 (.A(sigout[2]), .Z(sigout[3]));
INVDL inv_u004 (.A(sigout[3]), .Z(sigout[4]));
...
INVDL inv_u117 (.A(sigout[116]), .Z(sigout[117]));
INVDL inv_u118 (.A(sigout[117]), .Z(sigout[118]));
INVDL inv_u119 (.A(sigout[118]), .Z(sigout[119]));
INVDL inv_u120 (.A(sigout[119]), .Z(sigout[120]));

`else
assign sigout = { sigin, {60{!sigin, sigin}} };
`endif

Endmodule

```

Figure 8.0 Improved Delay Chain Verilog Code with IFDEF Construct

Now the delay chain is completely removed from the RTL simulation and VCS can optimize the run. Here's the profile output with recoding:

```
// Synopsys VCS 6.2R12
// Simulation profile: vcs.prof
// Simulation Time: 124.410 seconds
```

```
=====
TOP LEVEL VIEW
=====
```

TYPE	%Totaltime
PLI	1.15
VCD	0.02
KERNEL	13.34
DESIGN	85.50

```
-----
```

```
=====
MODULE VIEW
=====
```

Module (index)		%Totaltime	No of Instances	Definition
hurricane_tb	(1)	6.79	1	../tb/hurricane_tb_nodump.v:31.
dll_mux	(2)	5.90	1720	../rtl/ddrctlr/rtl/dll_mux.v:21.
pdisp	(3)	4.92	8	../rtl/hurricane/rtl/pdisp.v:33.
rctl	(4)	3.76	8	../rtl/hurricane/rtl/rctl.v:32.
dll_buf	(5)	3.66	1732	../rtl/ddrctlr/rtl/dll_buf.v:21.
xaux_regs	(6)	2.93	8	../rtl/hurricane/rtl/xaux_regs.v:249.
dll_delay_element	(7)	2.63	1720	../rtl/ddrctlr/rtl/
dll_delay_element.v:20.				
delaychain	(8)	2.08	56	../top/rtl/delaychain.v:15.
tdc_cdb	(9)	1.67	1	../rtl/tdc/rtl/tdc_cdb.v:16.
spsram_1536x32	(10)	1.32	8	/projects/clibs/rams_nobist_M4one/
0.15vst_2.0/verilog_fix/spsram_1536x32.v:8.				

Figure 9.0 New Profile with Recoding

The delaychain.v file has been removed from the list of top hitters. Note that the overall simulation time has decreased to 124.4 seconds, saving us more valuable simulation time.

5.3 Cleaning the pli.tab File

If you are using pli.tab files for debugging or inline of special C-based testing, look in the pli.tab files for anything that resembles the following:

```
acc:rw,cbka: *
```

Figure 10.0 PLI Resource Hog Example

The * means add the pli acc hook for every signal in the entire design, consuming valuable resources and simulation time. Be prudent and determine if you really need all that visibility into your simulation!

Many engineers opt to use a different debugging & waveform viewing tool than the built-in Virsim application. One of the leading tools, Debussy, ships a pli.tab file to use with the application so that VCS can map in the PLI routines for simulation debugging and waveform dumping. Unfortunately, Debussy ships a pli.tab file that is unoptimized.

```

$fsdbVersion check=plicompileNoArgs call=plitaskVersion
$fsdbDumpfile check=plicompileDumpfile call=plitaskDumpfile
♦ $fsdbDumpvars check=plicompileDumpvars call=plitaskDumpvars misc=plimiscFSDB
acc=read,callback_all:%*
♦ $fsdbDumpvarsToFile check=plicompileDumpvarsToFile call=plitaskDumpvarsToFile
misc=plimiscFSDB acc=read,callback_all:%*
$fsdbSwitchDumpfile check=plicompileSwitchFile call=plitaskSwitchFile
$fsdbAutoSwitchDumpfile check=plicompileAutoSwitchFile call=plitaskAutoSwitchFile
$fsdbDumpon check=plicompileNoArgs call=plitaskDumpon misc=plimiscDumpon
$fsdbDumpoff check=plicompileNoArgs call=plitaskDumpoff
$fsdbDumpflush check=plicompileNoArgs call=plitaskDumpflush
$fsdbDumplimit check=plicompileDumplimit call=plitaskDumplimit
$fsdbDumpStrength check=plicompileDumpStrength call=plitaskDumpStrength
♦ $fsdbDumpMem check=plicompileDumpMem call=plitaskDumpMem misc=plimiscDumpMem
acc=read,callback_all:%*
♦ $fsdbDumpMemNow check=plicompileDumpMem call=plitaskDumpMemNow misc=plimiscDumpMem
acc=read,callback_all:%*
$fsdbInteractive check=plicompileNoArgs call=plitaskInteractive
misc=plimiscInteractive
♦ $fsdbDumpEBC check=plicompileEBC call=plitaskEBC misc=plimiscFSDB
acc=read,callback_all:%*
♦ $fsdbDisplay check=plicompileDisplay call=plitaskDisplay misc=plimiscFSDB
acc=read,callback_all:%*
$fsdbSuppress check=plicompileSuppress call=plitaskSuppress
$fsdbTest call=plitaskTest
$fsdbDebug check=plicompileDebug call=plitaskDebug
$debussy check=plicompileDebussy call=plitaskDebussyVCS
$vericom check=plicompileVericom call=plitaskVericom
$db_breakatline check=plicompileLineBreak call=plitaskLineBreak
$db_breakonconcatline check=plicompileLineBreak call=plitaskOnceLineBreak
$db_deletelinebreak check=plicompileDeleteLineBreak call=plitaskDeleteLineBreak
$db_showlinebreak check=plicompileNoArgs call=plitaskShowLineBreak
$db_setfocus check=plicompileSetFocus call=plitaskSetFocus
$db_deletefocus check=plicompileDeleteFocus call=plitaskDeleteFocus
$db_enablefocus check=plicompileEnableFocus call=plitaskEnableFocus
$db_disablefocus check=plicompileDisableFocus call=plitaskDisableFocus
$db_stopatfocus check=plicompileNoArgs call=plitaskStopAtFocus

```

Figure 11.0 Unoptimized Debussy pli.tab file.

The problem with the standard Debussy PLI file is that some of the items defined use all of the PLI operations when they can be optimized down to just a task-based call. The items that can be optimized are marked above with a ♦ symbol. Here is a modified version of the same file that improved the performance of our debugging simulations by about 15% in runtime, just by changing the %* denotations with %TASK:

```

$fsdbVersion check=plicompileNoArgs call=plitaskVersion
$fsdbDumpfile check=plicompileDumpfile call=plitaskDumpfile
$fsdbDumpvars check=plicompileDumpvars call=plitaskDumpvars misc=plimiscFSDB
acc=read,callback_all:%TASK
$fsdbDumpvarsToFile check=plicompileDumpvarsToFile call=plitaskDumpvarsToFile
misc=plimiscFSDB acc=read,callback_all:%TASK
$fsdbSwitchDumpfile check=plicompileSwitchFile call=plitaskSwitchFile
$fsdbAutoSwitchDumpfile check=plicompileAutoSwitchFile call=plitaskAutoSwitchFile
$fsdbDumpon check=plicompileNoArgs call=plitaskDumpon misc=plimiscDumpon
$fsdbDumpoff check=plicompileNoArgs call=plitaskDumpoff
$fsdbDumpflush check=plicompileNoArgs call=plitaskDumpflush
$fsdbDumplimit check=plicompileDumplimit call=plitaskDumplimit
$fsdbDumpStrength check=plicompileDumpStrength call=plitaskDumpStrength
$fsdbDumpMem check=plicompileDumpMem call=plitaskDumpMem misc=plimiscDumpMem
acc=read,callback_all:%TASK
$fsdbDumpMemNow check=plicompileDumpMem call=plitaskDumpMemNow misc=plimiscDumpMem
acc=read,callback_all:%TASK
$fsdbInteractive check=plicompileNoArgs call=plitaskInteractive
misc=plimiscInteractive
$fsdbDumpEBC check=plicompileEBC call=plitaskEBC misc=plimiscFSDB
acc=read,callback_all:%TASK
$fsdbDisplay check=plicompileDisplay call=plitaskDisplay misc=plimiscFSDB
acc=read,callback_all:%TASK
$fsdbSuppress check=plicompileSuppress call=plitaskSuppress
$fsdbTest call=plitaskTest
$fsdbDebug check=plicompileDebug call=plitaskDebug
$debussy check=plicompileDebussy call=plitaskDebussyVCS
$vericom check=plicompileVericom call=plitaskVericom
$db_breakatline check=plicompileLineBreak call=plitaskLineBreak
$db_breakonceatline check=plicompileLineBreak call=plitaskOnceLineBreak
$db_deletelinebreak check=plicompileDeleteLineBreak call=plitaskDeleteLineBreak
$db_showlinebreak check=plicompileNoArgs call=plitaskShowLineBreak
$db_setfocus check=plicompileSetFocus call=plitaskSetFocus
$db_deletefocus check=plicompileDeleteFocus call=plitaskDeleteFocus
$db_enablefocus check=plicompileEnableFocus call=plitaskEnableFocus
$db_disablefocus check=plicompileDisableFocus call=plitaskDisableFocus
$db_stopatfocus check=plicompileNoArgs call=plitaskStopAtFocus

```

Figure 12.0 Optimized Debussy pli.tab file.

At the time of writing this paper, Debussy now supports a Direct Kernel Interface to VCS which is more efficient than using the PLI. If possible switch to the new Direct Kernel Interface model.

5.4 Using the +vcs+learn+pli Option

If your simulation heavily utilizes PLI routines, squeezing more performance out of the PLI interface is truly necessary. VCS incorporates a +vcs+learn+pli compile flag to learn which PLI items are being utilized in what mode during the simulation. VCS then produces a new pli.tab file, optimized for only the specific PLI access routines that can be used for future simulations.

In our test case, using the +vcs+learn+pli routine boiled down the accesses used into the learn_pli.tab file with a small number of PLI routine options specified:

```

//////////////////////////////// SYNOPSIS INC //////////////////////////////////
//                               PLI LEARN FILE
// AUTOMATICALLY GENERATED BY VCS(TM) LEARN MODE
////////////////////////////////////////////////////////////////////////
acc=rw:sync_sram
    //SIGNAL bt_baddr:rw
    //SIGNAL bt_data:rw
    //SIGNAL bt_eaddr:rw
acc=r:master64
acc=r:master32
acc=r:target32
acc=r:target64
acc=r:ucml5_pcixdrv
acc=rw:hurricane_tb
    //SIGNAL line:rw
    //SIGNAL string_clock:rw
    //SIGNAL real_time:rw
acc=rw:PP3PLPrtDm
    //SIGNAL chkInLine:rw

```

Figure 13.0 Generated pli.tab File

We incorporated this new `pli_learn.tab` file by using the `+applylearn` flag in conjunction with another compile. About a 10% speed improvement was realized in simulation runs. Keep in mind that if the PLI interface changes or a new PLI function is added, the `+vcs+learn+pli` option needs to be rerun or your simulation may quit working.

6.0 Performance Results

Cleaning up your simulation with the profiling analysis & compile/run script optimizations will do wonders for simulation speed improvement. In test case #1, we had a baseline, un-optimized compile and run script that looked as follows:

```

#!/usr/local/bin/perl -w
$plat = `uname -s`;
chomp $plat;
$ENV{PLI_LIB} = "/projects/Verilog_pli/lib/`uname -s`";
$vcs_cmd = "vcs -notice +acc+2 -I -Mupdate ";
$vcs_cmd .= "-P $ENV{DEBUSSY_PLI}/debussy.tab ";
$vcs_cmd .= "$ENV{DEBUSSY_PLI}/pli.a ";
$vcs_cmd .= "-P ./pli.tab ./pli.c +incdir+../tb ";
$vcs_cmd .= "+define+ARC_MYSRAM +define+BEHAVE ";
$vcs_cmd .= "+define+PLL_resolution_lps ";
$vcs_cmd .= "-P /projects/Verilog_pli/lib/${plat}/get_plusarg.tab ";
$vcs_cmd .= "-P /projects/Verilog_pli/lib/${plat}/value.tab ";
$vcs_cmd .= "/projects/Verilog_pli/lib/${plat}/libvalue.so ";
$vcs_cmd .= "+define+ARC_MYSRAM +define+PCI66 ";
$vcs_cmd .= "-P /projects/Verilog_pli/lib/${plat}/fileio.tab ";
$vcs_cmd .= " $ENV{PLI_LIB}/libget_plusarg.so ";
$vcs_cmd .= "$ENV{PLI_LIB}/libfileio.so ../lib/timescale.v ";
$vcs_cmd .= "+define+PLX -Mdir=../bin/csrc ";
$vcs_cmd .= "-CC \"-I $ENV{VCS_HOME}/include/ -DMUNIX -DUNIX\" ";
$vcs_cmd .= "/projects/hurricane/bin/onlineChecker.a ";
$vcs_cmd .= "-P /projects/Verilog_pli/lib/SunOS/rascalint.tab ";
$vcs_cmd .= "-larc-neutral-pli-Verilog -lrascalint ";
$vcs_cmd .= "-f filelist.v $copts -o simv ";
print "VCS Command : \n\n$vcs_cmd\n";
system("$vcs_cmd | tee compile.log");

```

Figure 14.0 Baseline Compile Script

Here is the breakdown of simulation time improvement across a suite of tests with incremental changes. Here are the column designations:

- A:** Baseline compile and run script with the delay chains enabled
- B:** Removal of instantiated gate-level delay chains
- C:** Removing `+acc+2`, `-I`, and `-PP`
- D:** Removing compile-in of Debussy PLI and other debugging PLIs which weren't used for regression simulations
- E:** With `+nospecify` compile switch

In columns B-E, each item builds upon the previous setting so column D has items from B, C and D. All times are measured in CPU seconds and were executed on single Linux P4 1.7GHz machine with 1.5 GB of system memory.

Test Name	A	B	C	D	E
ahb_cfg_test1	30.770	7.340	7.360	6.650	5.940
ahb_cfg_test2	13.130	4.370	4.380	4.170	3.740
ahb_cfg_test_incr	186.830	33.860	33.780	29.530	25.740
ahb_cfg_wr_rd	67.610	13.980	13.940	12.400	10.980
ahb_ddr_bw	115.140	22.310	22.250	18.970	16.680
ahb_ddr_test1	61.570	13.000	12.940	11.260	9.890
ahb_ddr_test_incr	103.870	21.420	21.270	18.510	16.320
ahb_ddr_wr_rd	100.100	20.520	20.340	17.400	15.400
ahb_memctl_test1	217.370	34.560	34.330	29.610	25.220
ahb_memctl_test_sdram	131.330	23.970	23.900	21.170	18.420
gmi_mission_test1	416.290	76.550	76.510	67.790	59.320
gmi_mission_test2	416.880	76.840	76.940	67.890	59.560
gmi_mission_test3	416.340	76.850	76.630	67.670	59.470
gmi_pause_test1	76.250	15.730	15.700	13.920	12.560
gmi_ser_test	97.630	18.230	18.180	16.160	14.140
Speed Increase over (A)	-	533%	534%	608%	693%

Figure 15.0 VCS Simulation Improvements, Delay Chain optimization

Note that each item incrementally improves the original time. Getting rid of the delay chain dramatically optimized the performance. Just by cleaning the delay chains that were hogging valuable CPU time resulted in over a 5X speed improvement! Let's look at the same results with the optimized Verilog code (B) as the baseline to show the improvements just by command-line switches alone in Figure 16.0.

Test Name	B	C	D	E
ahb_cfg_test1	7.340	7.360	6.650	5.940
ahb_cfg_test2	4.370	4.380	4.170	3.740
ahb_cfg_test_incr	33.860	33.780	29.530	25.740
ahb_cfg_wr_rd	13.980	13.940	12.400	10.980
ahb_ddr_bw	22.310	22.250	18.970	16.680
ahb_ddr_test1	13.000	12.940	11.260	9.890
ahb_ddr_test_incr	21.420	21.270	18.510	16.320
ahb_ddr_wr_rd	20.520	20.340	17.400	15.400
ahb_memctl_test1	34.560	34.330	29.610	25.220
ahb_memctl_test_sdram	23.970	23.900	21.170	18.420
gmi_mission_test1	76.550	76.510	67.790	59.320
gmi_mission_test2	76.840	76.940	67.890	59.560
gmi_mission_test3	76.850	76.630	67.670	59.470
gmi_pause_test1	15.730	15.700	13.920	12.560
gmi_ser_test	18.230	18.180	16.160	14.140
Speed Increase over (B)	-	0%	14%	30%

Figure 16.0 VCS Simulation Improvements, Command-Line Switches

Pruning the compile and run scripts gained an average of 30% simulation speed performance!

7.0 Speeding up Gate Simulations

Gate simulations are useful in verifying the final product acts like it is supposed to function in a full-timing manner. They are very useful in verifying that initialization and reset of the design is correct, no hidden multi-cycle paths got lost at static timing analysis, and that timing issues are clean at process/voltage/temperature corners. Gate simulations are very large and slow, as they incorporate back-annotated layout timing data along with library timing models into the chip model. The last 20 million gate project at Corrent had gate sims taking 8 hours to compile and over 24 hours of run time for one simulation occupying a full 5 GB of system memory. Many engineers limit how many gate simulations are done purely due to resources and timing. This is also a good area for a speed optimization.

7.1 VCS Command Line Switches

For gate level simulations there are two switches that can be utilized to speed performance:

- +memopt
- +timopt

Both flags must be utilized at compile time as compile options.

7.2 The +timopt Flag

The `+timopt` timing optimizer can yield large speedups for full-timing gate-level designs. VCS makes its optimizations based on the clock signals and sequential devices that it identifies in the design. The `+timopt` flag is particularly useful when using SDF files, since SDF files can't be optimized with the Radiant Technology `+rad` flag. You enable `+timopt` with the `+timopt+[clock_period]` compile time option, where the clock period argument is the shortest clock period (or clock cycle) of the clock signals in the design. For example:

```
+timopt+100ns.
```

This option specifies that the shortest clock period is 100ns. In the hurricane testcase, `+timopt` was able to automatically optimize 32% of the sequential elements in the design. A configuration file is also written out which enables the user to manually specify other potential sequential elements for further optimizations.

7.3 The +memopt flag (What if My Design Won't Compile?)

If there is a very large module definition in a flat gate-level design, memory consumption could be very high. One possible solution to reduce memory usage is the `+memopt` compile time option. This option enables a number of optimizations that reduce the amount of memory needed during compilation. There may be a runtime performance cost using this compile time option depending upon your design.

If you find that you are using up too much memory even when utilizing the `+memopt` option, you could be reaching the per process memory limits imposed by the system on the compilation process. Another switch to try is the `+memopt+2` compile-time option. The `+memopt+2` option spawns a second process for some of the memory optimizations. The compilation log file will contain entries if `+memopt+2` optimizations occurred. Be sure to check the log file after compilation.

7.4 Process Size and Limitations

Keep in mind that various computing platforms impose different process size limitations. On Solaris 32-bit machines, a process is limited to 4GB in memory utilization. On Solaris 64-bit machines, the sky is the virtual limit depending upon how much real physical memory and swap space is present. One of our large 64-bit Solaris servers contains 24 GB of physical memory and a 100 GB swap partition just for those rare occurrences.

On our Linux compute farms, there are a small group of Linux machines that contain 4GB of physical memory. Out-of-the-box generic Linux imposes a 3GB process size limitation. Edit the `/usr/src/linux-2.4/include/asm-i386/page.h` file and change `0xC0000000` to `0xEC000000` to bump this up to 3.7 GB of process size. This was highly useful for those compiles that did not crest the top of memory limit but were large enough to go past the 3GB process size.

7.5 Cleaning Libraries

Libraries can be inefficiently written just as bad as design code. One of the biggest items to watch out for is embedded “# delays” such as section 3.2.1, that can still creep in.

Consider the following code for module FD2:

```
module FD2(d,cp,cd,q,qn);

    input d,cp,cd;
    output q,qn;

    reg q;
    assign qn=~q;
    always@(posedge cp or negedge cd)
        q = #5 cd ? d : 1'b0;
endmodule
```

Figure 17.0 Initial FD2 Module

Aside from the more difficult to read conditional assignment with the #5 delay in it, these modules are pretty much the same. The second works properly without the “#” delay (no race condition) and is more easy to read:

```
module FD2(d,cp,cd,q,qn);

    input d,cp,cd;
    output q,qn;

    reg q;
    assign qn=~q;
    always@(posedge cp or negedge cd)
        if(cd == 0)
            q <= 1'b0;
        else
            q <= d;
endmodule
```

Figure 18.0 Cleaned FD2 Module

Another item to scan in libraries are library cells or Verilog modules that are nearly the same but *not* quite identical. VCS can use an optimization (in concert with +rad) called ‘vectorization’. Basically, when Verilog modules look alike, VCS can take advantage of that fact. This was more of an issue in older versions (5.X) of VCS, however when coupled with +rad, these module similarities can have an impact on the design and where time is spent.

```

module lfsr_leaf1(d,clk,reset,q);

    input d,clk,reset;
    output q;
    wire q;
    wire q_unused;

    EN en_1(d,q7,xout);
    FD2 fd2_1(xout,clk,reset,q1,q1b),
    fd2_2(q1,clk,reset,q2,q2b),
    fd2_3(q2,clk,reset,q3,q3b),
    fd2_4(q3,clk,reset,q4,q4b),
    fd2_5(q4,clk,reset,q5,q5b),
    fd2_6(q5,clk,reset,q6,q6b),
    fd2_7(q6,clk,reset,q7,q7b),
    fd2_8(q7,clk,reset,q8,q8b),
    fd2_9(q8,clk,reset,q9,q9b),
    fd2_10(q9,clk,reset,q_unused,q10b);

    assign q = ~q10b;

endmodule

```

Figure 19.0 'lfsr_leaf1' code

```

module lfsr_leaf2(d,clk,reset,q);

    input d,clk,reset;
    output q;

    EN en_1(d,q7,xout);
    FD2 fd2_1(xout,clk,reset,q1,q1b),
    fd2_2(q1,clk,reset,q2,q2b),
    fd2_3(q2,clk,reset,q3,q3b),
    fd2_4(q3,clk,reset,q4,q4b),
    fd2_5(q4,clk,reset,q5,q5b),
    fd2_6(q5,clk,reset,q6,q6b),
    fd2_7(q6,clk,reset,q7,q7b),
    fd2_8(q7,clk,reset,q8,q8b),
    fd2_9(q8,clk,reset,q9,q9b),
    fd2_10(q9,clk,reset,q,q10b);

endmodule

```

Figure 20.0 'lfsr_leaf2' code

By coding the two leaf modules identically, less time is spent doing port evaluations in the VCS kernel and more time is spent evaluating the design (approximately 2% of a 25 second run).

While not much code is in this example, consider nearly 50 minutes of a 24-hour run and multiply that time savings by many many modules across a large SOC design!

The Synopsys RTL rule checking tool "LEDA" includes rulesets for not only synthesis, but for VCS performance optimizations as well. This can be used to screen libraries and Verilog code for performance as well as potential race conditions.

8.0 Summary

In summary, the heavy performance items to prune from compile and run scripts are:

- -I
- +cli

- `+acc+2`
- `-P [library]`
-

The items to ensure RTL sims run efficiently are:

- `-Mupdate -o csrc` (use local disk)
- `+rad`
- `+nospecify`
- Optimized `pli.tab` files
- `+nbaopt` (if required)
- Good Verilog coding practices

Clean your Verilog code and libraries of any #0 or #1 delays and proof your code for style problems. Profiling your sim with the `+prof` option gives the user a heads-up into time hogs. Using the `+timopt` can increase gate-level simulation performance and using the `+memopt` may give you that extra bit of headroom to compile gate simulations. Clean your PLI `.tab` files by correcting the * options and when possible use the `+vcs+learn+pli` option to optimize the PLI calls.

Straight out of the box, VCS is a powerful simulator but needs to be customized to get the most performance based upon the user constraints. Not all optimization techniques are obvious, but with this paper and little bit of trial and error, your simulations can be running quicker and more efficient. Our large designs at Corrent yielded up to a 6x performance improvement by following these tips!

The authors would like to thank Mark Warren, Synopsys Corporation, for his review and comments of the paper and that fruit basket which fueled the ambition to write this paper. This paper, along with other SNUG papers the author has written, can be found on <http://gateslinger.com/chiphead.htm>.

9.0 References

VCS 5.0 and 6.0 User Guides, Synopsys Corporation, 2002.

Test Benches: The Dark Side of IP Reuse, Gregg D. Lahti, San Jose SNUG 2000 paper.

<http://gateslinger.com/chiphead.htm> or

http://www.synopsys.com/news/pubs/snug/snug00/lahti_final.pdf

Verilog Nonblocking Assignments With Delays, Myths and Mysteries, Cliff Cummings, Boston SNUG 2002 paper. <http://www.sunburst-design.com/papers/>.

ESNUG posts: 380 item 11, 383 item 9, 387 item 16. <http://deepchip.com/esnug.html>

Solvnet: <http://solvnet.synopsys.com>